Contract-Violation Handlers

Document #:	P2811R2
Date:	2023-4-20
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

Numerous use cases for contracts in production environments depend upon handling contract violations in a consistent and locally defined way. Based on existing designs deployed at scale over many years, we present here a proposal to allow for link-time customization of contract-violation handling, along with examples of how this method might satisfy a wide variety of important, practical, and well-known usage scenarios.

Contents

Revision History

Revision 2

- Removed the `ignore` semantic enumeration
- Added discussion of counting violations for observed violations
- Added section on safe stopping
- Explicit clarification on proposed modification added to introduction

Revision 1

- Clarifications on the undefined behavior when using `longjmp`
- Reasoning for the `contract_semantic::ignore` and `contract_violation_detection_mode::unknown` enumerators
- An explanation for why the violation handler is in the global namespace and clarification that it should be attached to the global module
- Other minor corrections

Revision 0

- Original version of the paper for discussion during an SG21 telecon

1 Introduction

C++20 Contracts, prior to their removal, describe a conditionally supported mechanism for installing a user-supplied contract-violation handler. This callback function — invoked immediately upon each detected contract violation — was intended to support both (1) the handling of user-defined reporting of a contract violation, e.g., via what mechanism and in what format to report the error, and (2) managing the *semantics* of contract violations within the program, e.g., terminate (the default), long jump (save and quit), throw (and what to throw), or, in some use cases, continue as if contracts were disabled.

Handling contract violations via a user-provided callback is an established, well-tested approach that is deployed in many modern assertion facilities. In particular, this approach has evolved directly from that used by BDE[1] in `bsls_assert` and `bsls_review`. User-provided contract-violation handlers were first deployed to production at Bloomberg in 2004 and have been in continuous use ever since. In addition, this approach has had later versions make their way through both LEWG and EWG to be moved for standardization[2] only to be rejected or removed at plenary prior to Standard publication. That is, the eventual lack of acceptance of a Contracts facility for C++17 and again for C++20 had nothing whatsoever to do with the utility or design of the contract-violation handler; instead, the issue was the design as a whole, i.e., the intrinsic use of macros or perceived design instability.

Attempting to clarify the semantics of contract checks[3] made clear that the local aspects of what a contract check should do — i.e., determining whether control flow can continue normally after a

[1] See [bde14].
[2] See [N4378] and [P0542R5].

violation and whether the contract is checked — are separate from the decision of how precisely to report a contract violation.

The current MVP[4] has no mechanism for altering the behavior on contract violation, nor does it provide much guidance on what default behavior should occur when a contract violation is detected. The only mandatory behavior dictated by the `Eval_and_abort` build mode, where all contract-checking annotations are given the *enforce*[5] semantic, is program termination.

We propose here two modifications to the contracts MVP.

1. Add the ability to provide a function — the contract-violation handler — to be invoked as part of the contract-violation handling process. In a build mode such as `Eval_and_abort`, program termination will occur when the contract-violation handler returns normally.
2. Establish a recommended practice for the behavior of the default contract-violation handler that platforms might wish to adopt when applicable.

Concretely, the current MVP makes the behavior on contract violation in `Eval_and_abort` mode implementation-defined, only requiring that the program does terminate on such violations (instead of throwing or invoking `longjmp`). Given this assertion (using the syntax of [P2521R3]):

```
ASSERT( X );
```

The current behavior in `Eval_and_abort` mode could be imagined to be transform into something like this:

```
if (X) {} else {
   __invoke_platform_violation_handling();
   __terminate_program();
}
```

Here the `__invoke_platform_violation_handling()` intrinsic is expected to not `throw` or invoke `longjmp`, but otherwise has implementation-defined behavior.

This proposal makes this implementation-defined behavior into fully specified behavior, requiring that the compiler prepare an appropriately populated `std::contracts::contract_violation` object and then pass it to the (possibly) replaceable function `::handle_contract_violation`.

We do not propose removing the program termination on normal returns from handling a contract violation, as that would allow the *observe* semantic which should be the subject of a future proposal incorporating additional build modes and meta-information with which to control the use of that contract semantic.

2 Motivation

Custom violation handlers turn an overly simplified Contracts facility that is highly ineffective in many environments into a moderately flexible and practicable one. Importantly, primary control

[3] See [P1332R0], [P1429R3], and [P1607R1].
[4] See [P2521R3].
[5] The *enforce* semantic for contract violations was originally named `check_never_continue` in [P1332R0] and is identical, other than the name, to the semantic originally described there.

of the semantics of a contract check — i.e., whether it is checked and how control flows when a violation is detected — remain governed by the choice of build mode. In the `No_eval` build mode, contracts will continue to be *ignored*, and no contract-violation handler will be invoked. The `Eval_and_abort` is currently the only build mode that would (or could) invoke the contract-violation handler immediately following the detection of a contract violation. If, after a contract violation is detected, the contract-violation handler returns normally, program execution will be terminated. A typical custom violation handler will log appropriately; after that, the only mechanisms to circumvent program termination are to throw an exception, enter an infinite loop, or invoke `std::longjmp`. Continuation in this build mode is simply not an option.

Throughout the standardization process, two points have become abundantly clear: (1) No one specification of violation handling is correct for all users on all platforms, and (2) having a consistent and mostly portable way to customize behavior will greatly increase the utility (and thus, in some views, the viability) of a Contracts facility for a wider range of users. Executing any code when a contract violation is detected is often a risk because one can never be certain that the program is not already exhibiting undefined behavior due to earlier bugs that were not detected by an appropriate contract check. This risk must be balanced with the beneficial utility that can be provided by producing useful diagnostics or making attempts at recovery — or at least saving the user's data — in an appropriately structured manner. We assert that having the ability to customize contract-violation handling beyond just logging is, generally speaking, necessary and almost always a net positive.

On some very specialized platforms, arbitrary selection of a contract-violation handler might be considered an unacceptable security risk. Therefore, the ability to provide a replacement contract-violation handler is only *conditionally supported*. We suggest that providing a candidate function for replacement on platforms that do not support such replacement be an error so as to minimize any related confusion. Platforms having somewhat less stringent security concerns might choose to disallow replacement and instead to provide a choice of violation-handling implementations from a selection of well-vetted, vendor-provided routines. Again, we would expect that on general-use platforms, developers will be able to create and supply fully custom violation-handling routines at build time.

3 Proposal

Our proposed solution to broaden the viability of a still-minimal Contracts facility starts with how to define the object that will capture the details of a contract violation, how to specify a contract-violation handler, and what the recommended default contract-violation handler should do.

3.1 An Extensible `contract_violation` Value Type

Proposal 1: The Standard Library `std::contracts::contract_violation` Type

A new language-support type, `std::contracts::contract_violation`, will be added to the Standard Library and will be designed for extensibility in an ABI-compatible manner.

To specify a custom violation handler, we will need to provide a type to represent the information

that will be gathered and made available to a contract-violation handler when a contract violation occurs. This type might have any number of properties, but we want to ensure that it can evolve while remaining ABI compatible.

We therefore recommend the following expectations and requirements for this `contract_violation` type.

- Property types will be builtin types, enumerations, or standard-library value types such as `std::source_location`. Strings will be null-terminated byte strings denoted by a `const char*`.[6]
- Each property accessor will return by value, not reference. Hence, a `contract_violation` object itself is never required to maintain a member for any of these properties unless it chooses to do so.
- Each property of `std::contracts::contract_violation` objects will have a recommended practice for any values supplied when a contract violation is detected but no requirements as to the specific values that property must be populated with. Implementations may, therefore, choose to store more information for improved diagnostics or instead carry less information in an executable for reduced overhead, potentially leaving that choice to the user.
- Objects of type `std::contracts::contract_violation` will be passed by `const&`, not by value, to the contract-violation handler.

Proposal 1.1: The `<contract>` Header

The type `std::contracts::contract_violation` shall be defined in a new language support header `<contract>`.

The C++20 Contracts facility has the primary declaration of this type in a new header, `<contracts>`, and we see no compelling reason to place it elsewhere. But, much like other complex systems (e.g., `std::pmr`), we now understand that this facility will inevitably evolve to require many other such supporting types; hence, we propose to give our new Contracts facility its own namespace under `std`, `std::contracts`. (This sort of logical-physical cohesion is considered by many to be an industry best practice.)

Proposal 1.2: The `location` Property

The type `std::contracts::contract_violation` shall provide this accessor:

```
std::source_location location() const noexcept;
```

The recommended practice for the `location` property is to provide a source location for identifying the contract-checking annotation that has been violated. When possible on a precondition, this property would ideally be the source location of the point of function invocation. When the invocation location cannot be ascertained and on other contract checks, the location provided would be the source location of the contract check itself.

[6] See [P1639R0] for the LEWG reasoning behind the use of `const char*` in `std::source_location` and associated arguments in favor of using `const char*` over `std::string_view` in types necessary to the use of core language features such as `contract_violation`.

Compiler flags that request that built executables not store information regarding the source code that produced the executable should not be rendered nonconforming by the need to populate this location property. To allow such options, producing a default-constructed `source_location` from this property would be permitted, although this option would be nonoptimal for those users who did not explicitly choose to have this information made unavailable.

Proposal 1.3: The `comment` Property

The type `std::contracts::contract_violation` shall provide this accessor:

```
const char* comment() const noexcept;
```

The recommended practice for the `comment` property is to include a textual representation of the predicate expression in the contract-checking annotation that has been violated. When storing the text of all potentially violated contract checks in a program is deemed to be too inefficient or cumbersome, returning the empty string (`""`) instead is recommended.

Proposal 1.4: The `detection_mode` Property

The `<contract>` header shall provide an enumeration having members contingent on which forms of violation detection are accepted:

```
namespace std::contracts {
enum class contract_violation_detection_mode {
  unknown,
  predicate_false,
  predicate_exception,
  predicate_detected_undefined_behavior
};
}
```

The type `std::contracts::contract_violation` shall provide this accessor:

```
contract_violation_detection_mode detection_mode() const noexcept;
```

Detection of a contract violation can be triggered in three ways. Typically, the violation will be triggered when a contract-checking predicate evaluates to `false`, thereby clearly indicating that the expected boolean condition failed to hold. In such cases, the value returned from the `detection_mode` accessor will be `contract_violation_detection_mode::predicate_false`. The various subproposals of Proposal 3 in [P2751R0] indicate that other situations, such as a predicate that throws or that has readily detectable undefined behavior, are also potential candidates for triggering a contract violation, and in such cases, indicating which mechanism led to the violation can greatly help guide the violation-handling behavior.

To retain the implementation flexibility needed to clearly invoke the contract-violation handler when undefined behavior that is not necessarily associated with a contract-checking annotation occurs, we also include the `unknown` enumerator to allow for a mechanism to communicate that a violation is being handled that was triggered from an unspecified or implementation-defined mechanism.

Proposal 1.5: The `semantic` Property

The `<contract>` header shall provide this enumeration:

```
namespace std::contracts {
enum class contract_semantic {
  enforce
  /* To be extended with implementation-defined values and by future standards */
};
}
```

The type `std::contracts::contract_violation` shall provide this accessor:

```
std::contracts::contract_semantic semantic() const noexcept;
```

Contract checking proposals, including the MVP, hinge on selecting, at build time, a runtime semantic controlling how detection and enforcement of each individual contract-checking annotation will be performed. Our research and experience tells us that at least four well-defined semantics are sound and practical to apply in appropriate circumstances. As of now, the MVP allows for contracts having two of those semantics, i.e., *ignore* and *enforce*, which are selected for all contract-checking annotations within a program based on the build modes `No_eval` and `Eval_and_abort` respectively. In effect, this choice will result in users observing the results on only *enforced* contract-checking annotations in a contract-violation handler. Therefore we can forgoe even specifying the `ignore` enumerator as it would currently go unused. We hope that future evolution, however, will result in satisfying the use cases of those who wish for contract checks having other checked semantics, such as *observe*, since knowing which semantic is in effect for a particular violation greatly aids in programming the violation handler to behave properly.

In particular, when a contract is enforced, we know that the program will not continue normally after invoking the handler. A handler that wishes to proceed in any way other than program termination, such as by throwing or invoking `longjmp`, would thus need to do so after logging and prior to returning normally. But consider a future kind of contract-checking statement whose only two viable semantics were *observe* and *ignore*. Hard coding the throw or long-jump into the hander would interfere with the intent of always continuing but sometimes logging. In that case, we would want to do the long-jump or throw only when the contract check had the *enforce* semantic.

Moreover, since continuation might result in many violations of the same contract-checking annotation, a robust violation handler would not necessarily want to attempt to log a message on every violation. In general, the diagnostic of the first violation of each contract is very helpful and, in many cases, only one violation might occur, so skipping that diagnostic entirely is ill advised. Diagnostics of repeated violations quickly become unhelpful, so best practice is to employ some form of exponential backoff for logging. This backoff strategy requires the violation handler to count violations of each *observed* contract check in a safe and reasonably performant manner. Providing a light stack trace to see the entire call chain is another useful technique, especially in enforce mode.

In practice, a `contract_violation` whose semantic has the `ignore` value will never occur; if we are not evaluating the contract-checking annotation's predicate to determine if there is a violation, a contract-violation handler will never be invoked. The *ignore* semantic is still, however, one that the MVP allows contracts to have, and thus we could include it, but with no current programmatic

need for it we omit it for now.

Proposal 1.6: The `kind` Property

The `<contract>` header shall provide this enumeration:

```
namespace std::contracts {
enum class contract_kind {
  pre,
  post,
  assert
};
}
```

The type `std::contracts::contract_violation` shall provide this accessor:

```
std::contracts::contract_kind kind() const noexcept;
```

Whether the contract annotation violated was a *precondition*, *postcondition*, or *assertion* might guide the form of logging statements produced by a custom contract-violation handler.

Future extensions to Contracts might add new forms of contract checks, such as procedural interfaces[7] or class invariants, which could then be distinguished in a contract-violation handler by producing distinct values of the `contract_kind` enumeration.

3.2 Contract-Violation Handler

Proposal 2: Contract Violations Invoke a (Potentially) Replaceable Function

When a contract violation is detected, prior to other specified behavior that is associated with the contract annotation, a function named `::handle_contract_violation` that is attached to the global module will be invoked. This function

- may be `noexcept`,
- may be `[[noreturn]]`,
- shall return `void`, and
- shall take a single argument of type `const std::contracts::contract_violation&`.

Whether this function is replaceable is implementation defined.

Not all platforms, especially those that seek to have a thoroughly auditable security-conscious deliverable, will want to support a replaceable[8] contract-violation handler. We therefore propose that whether `handle_contract_violation` is replaceable be unspecified. For platforms where replaceability is not supported, defining the function shall lead to a link error (a clear and early indication to users who attempt to replace `handle_contract_violation` that they will not be able to take advantage of this portion of the contracts feature). Platforms that do not allow for replacement of the violation handler are nonetheless encouraged to instead provide to users at build time a finite set of alternative behaviors that the default violation handler may have.

[7] See [P0465R0].

[8] This proposal adds `::handle_contract_violation` to the set of replaceable functions the Standard defines, which currently includes various overloads of the global `operator new` and `operator delete`. Just like those

Notably, the language and Standard Library provide no mechanism to alter the behavior of the contract-violation handler at run time. A general feature having that purpose has been deemed by some to be too large a security risk on many platforms, though a custom violation handler can be crafted to achieve that effect.

3.3 Default Violation Handler

Proposal 3: Default Violation-Handler Behavior

Recommended practice is that the default violation handler will output diagnostic information describing the pertinent properties of the provided `std::contracts::contract violation` object.

When the violation handler is not replaceable or when no replacement is provided, recommended practice is to provide a violation handler that outputs useful diagnostic information (such as the contents of the `std::contracts::contract_violation` object) to a standard error-reporting channel for the platform (such as `stderr`).

Looking forward, should the committee adopt the ability to *observe* contract violations,[9] recommended practice would be that the default violation handler log diagnostics only infrequently, such as with some form of exponential backoff counter. Logging a diagnostic for each repeated failure of the same contract-checking annotation can quickly down a system, and observation is intended to avoid exactly that problem.

Platform capabilities, limitations, and other concerns will, of course, lead to default violation handlers that do much more or much less. This behavior can range from launching a debugger to doing nothing at all.

4 Usage Examples

Providing just the hook of a custom violation handler effectively supports many use cases that would otherwise not be implementable using the MVP.

4.1 Custom Diagnostic Output

The most common use case for custom contract-violation handlers is to log the error to a particular output API in a particular format:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  std::cerr << "Contract violated at:" << violation.location() << std::endl;
}
```

functions, the violation-handler function is placed in the global namespace and attached to the global module.

[9] By *observe*, we mean "determine if a contract violation is detected by a specific contract-checking annotation and then, when a violation occurs, invoke the contract-violation handler and, if the handler returns normally, continue execution immediately following the contract-checking annotation." See [P1607R1] and the earlier [P1332R0], which referred to this behavior as `check_maybe_continue`.

Taking into the account the risks and potential rewards, a handler might choose to add stack traces, the time, some subset of static program state, or other information before allowing the program to terminate.

Other programs might want to provide feedback to a user more directly:

```
#include <windows.h>`
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  MessageBox(NULL,
             (LPCWSTR)L"Contract violation, abort?",
             (LPCWSTR)L"Contract violation",
             MB_OK);
}
```

On other platforms, a program might send messages to `syslog`, use other central logging APIs, store an event in a diagnostic-event recording system, or perform other environment-specific actions to record the detection of a contract violation.

4.2 Throw on Contract Violation for Recovery

Rather than aborting, an application might instead choose to handle all contract violations as exceptions by throwing a known contract-violation-exception type from the handler:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  throw my::contract_violation_exception(violation);
     // copies relevant fields from violation
}
```

Properly supporting this use of contract violations requires code be written with exception safety in mind, which also goes a long way toward supporting the use case of applications that do not have the choice to terminate.[10]

Since we, in general, do not want to make changes to the abstract machine itself, this use of exceptions does nothing to prevent termination related to `noexcept` functions further up the stack, but see Section 5 for more discussion on this topic.

Stack unwinding itself might, however, lead to other contract violations in the destructors of automatic variables on the stack, showing again that significant risk is involved in using exceptions as part of handling contract violations. Those who choose to do so must carefully evaluate the software they write to be sure it will handle such problems properly.

4.3 Propagating Predicate Exceptions

The proper response to an exception being thrown from the evaluation of a contract-checking annotation's predicate expression might arguably depend on context. Some applications might have resource recovery mechanisms to continue executing properly when, say, `std::bad_alloc` is thrown,

[10] See [P2698R0].

while others might have no viable way to recover from a logical error that was expressed as a thrown exception (a common if unfortunate practice, such as through the use of `std::vector::at`).

In most cases when a predicate fails to evaluate cleanly to something contextually convertible to `true`, something is amiss and is causing the contract-violation handler to be invoked. For those tasks that are capable of recovering from a thrown exception, we can easily have a contract-violation handler propagate the exception that escaped the evaluation of the predicate:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  if (violation.detection_mode() == detection_mode::predicate_exception) {
    throw;  // rethrow the current exception
  }
}
```

In other cases, users can inspect the particular exception that was thrown and propagate only those that are known to represent resource acquisition failures from which the application is designed to recover:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  // First, log appropriately.
  if (violation.detection_mode() == detection_mode::predicate_exception) {
    std::exception_ptr exception = std::current_exception();
    try {
       std::rethrow_exception(exception);
    } catch (const std::bad_alloc&) {
       throw; // rethrow bad_alloc
    } catch (...) {
       // Return normally to abort on other exception types.
    }
  }
}
```

4.4 `longjmp` for Recovery

One option to avoid aborting without throwing an exception is to use `std::longjmp` to move control flow up the stack to a primary event loop and begin recovery. The hard edge on this approach is that behavior is undefined if the destructors of any non-trivial automatic variables would need to be invoked when unwinding the stack from the invocation of `longjmp` to the invocation of the corresponding `setjmp`, i.e., if the pair of calls were replaced with a `try` and a `catch`.

This undefined behavior manifests in at least two distinct ways. Many platforms simply skip any of the corresponding destructors, leading to possible resource leaks (or worse, an imbalance in the invariants that an RAII object, such as a `std::lock_guard`, was intended to enforce) but then continuing execution from the point of invocation of `setjmp`. Other platforms,[11] however, will, when `longjmp` is invoked, unwind the stack in a manner very similar to what would happen when an exception is thrown. On platforms where `longjmp` will unwind the stack, all the same pitfalls that apply to attempting to recover via throwing an exception, will apply — although it can be expected

that even when crossing the boundary of a function with a nonthrowing exception specificattion unwinding will continue and `std::terminate` will not be invoked.

Using this `longjmp` approach requires carefully managing threads of execution and their currently associated `jmp_buf` instances, handling the platform-specific behavior when unwinding goes past non-trivial destructors, and overall structuring an entire application to be prepared to recover in this manner. The solution proposed in [P2784R0] is very similar to this approach in spirit, with similar associated benefits and potential pitfalls.

4.5 Performing a Safe Stop

For many purposes program termination is the safest approach to handling software defects — users will be notified of problems through the auspices of the infrastructure that executed the program, actions can be taken to recover at a higher level that has not suffered from unknown defects, and normal computing activity can resume productively.

Some uses, however, do not have the surrounding infrastructure or must avoid making the same level of demands of a user to make progress. Naive users happily executing a graphical interactive program will be unlikely to be served well by a program that simply disappears from their desktop for reasons they cannot readily identify. An embedded system navigating a car on a highway must, as a whole, continue controlling the car until it is in a safe situation to stop execution — the currently sleeping human being behind the wheel of the car will certainly appreciate that more than being given control of a speeding car unexpectedly on a busy highway.

Both of these extremes, and many in the middle, can benefit from a custom contract-violation handler being able to begin executing complex logic to *wind down* the system to a state where it is safe to stop. Recovery of the original processing goals might not be viable, but a minimal set of functionality can be launched to continue execution until a safe stopping point is reached.

For the client of software with a primary graphical interface this can be as simple as restarting the graphics-processing event loop within the violation handler in order to present an error dialog to the user before gracefully shutting down. This minimal runtime state can even give a user the ability to decide intelligently on what information to retain or discard before finally terminating the program.

A self-driving car, on the other hand, can run much more simplified and well-tested code paths when the only goal is to bring the vehicle to a halt on the shoulder of the road, out of the way of passing traffic and ready to wait for the surprised, groggy, yet still alive driver to manually take control of the vehicle once again.

All of this functionality is well-defined and actionable using a custom contract-violation handler.

4.6 Runtime-Selectable Violation Handler

Enabling runtime selection of the mechanism for violation handling provides an attack vector to malicious actors that are capable of both (1) updating that mechanism to arbitrary functions determined by the attacker and (2) forcing a contract violation to be detected after that. Many of those who deploy C++ software, however, might determine that an attacker capable of those two steps is likely capable of many other malicious acts, so the risk of enabling runtime selection is

[11] Such as MSVC; see https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/longjmp.

acceptable. Put another way, locking the door to the garage doesn't help if the attacker is already in your house and just makes unloading your groceries from your car more difficult.

A custom violation handler that simply delegates to a user-specifiable violation handler that can be altered at run time is straightforward to implement by maintaining a pointer to a violation-handling function with static storage duration that can be updated by clients as well as accessed by the violation handler:

```
class RuntimeViolationHandler
{
public:
  using Handler = void(*)(const std::contracts::contract_violation& violation);

private:
  static std::atomic<Handler> s_handler

public:
  static void invokeViolationHandler(const std::contracts::contract_violation& violation)
  {
     Handler handler = s_handler.load();
     handler(violation);
  }
  static void setViolationHandler(Handler handler)
  {
     s_handler.store(handler);
  }
  static void defaultViolationHandlerconst std::contracts::contract_violation& violation);
      // log details to stderr
};

std::atomic<RuntimeViolatinHandler::Handler>
  RuntimeViolationHandler::s_handler = &RuntimeViolationHandler.defaultViolationHandler;

void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
    RuntimeViolationHandler::invokeViolationHandler(violation);
}
```

Runtime selection of the contract-violation handling behavior enables several common use cases whose benefits depend on the environment in which they might be used:

- Choosing violation handling behavior based on configuration options not identified until after `main` has begun
- Dynamically altering what information is gathered by a violation based on what phase a program is in during its execution, i.e., storing and attempting to save user information only after the user login process has completed and normal program usage is underway
- Dynamically altering the form of recovery that might be attempted on violation, such as attempting to recover via thrown exception only after the main program execution loop is underway

Any custom violation handler could certainly modify its behavior based on program state. A generic runtime-replaceable facility such as this one allows for a library solution to leave the choice of specific violation-handling mechanics to a higher-level component.

4.7 Negative Testing of Non-`noexcept` Functions

For non-`noexcept` functions (though see Section 5), a runtime-selectable violation handler (such as the one implemented above) allows for effective, practical *negative testing* of contract checks in a fully well-defined manner. Negative testing is a tool to verify that contract checks have themselves been written to properly detect inputs outside of the domain of a function. The negative-testing algorithm consists of a few steps.

- Execute negative tests only in a build mode where the contract checks under test will be evaluated and invoke the custom violation handler, such as `Eval_and_abort` mode on a platform that allows replacing the violation handler.
- Prior to beginning a negative test, install a violation handler that will throw a known testing-only exception containing the violation details.
- Inside a `try...catch` block, execute the function with inputs that are out of bounds.
- The precondition checks on the function should detect the out-of-bounds input and lead to an immediate exception, unwinding the construction of the function arguments and being caught by the caller. Importantly, this method is the only one that doesn't leak when resource-allocating objects are passed by value.
- The caller then checks that control flow passed through the `catch` and the function under test did not return normally. Returning normally from the function or throwing any other exception type is considered a test failure.
- The caller also checks that the violation handler was invoked from a contract-checking annotation in an appropriate-seeming `location`. A violation in a different function called from within the function under test would indicate that the precondition was not properly checked.
- Once all tests are run, restore the runtime-selectable violation handler to the value it had before the negative test began.

4.8 Counting Repeated Violations

Should the Contracts facility evolve to enable contracts having the *observe* semantic it will quickly become possible for a single contract to be violated a significant number of times in a single program. Allowing violation handling to throw, and thus potentially return again to the same point of violation at a later point in execution, also opens up this possibility.

The overhead of producing a diagnostic for each violation when attempting to simply observe the violations that are otherwise benign, can be too large to enable normal business operations. On the other hand, accidentally skipping diagnostics for all occurrences of a particular contract check failing might hide real program defects, throwing the proverbial baby out with the bath water.

To fix these issues, a violation handler can keep a count of how many times a particular contract check has been violated and only emit a diagnostic for progressively fewer and fewer of them,

usually with a strategy such as only logging when the count is a power of 2, an approach known as *exponential backoff*.

The hard part of performing this type of backoff hinges on the act of identifying individual contract checks. The `source_location` provided on the `contract_violation` object is a rough approximation, but is somewhat lacking:

- For contracts on a templated entity the distinct instantiations of the template *might* be captured by the `function_name` property on the `source_location`, but many compilers choose to not keep the full text of the function name available to runtime violation handling, so this might be limited.
- Any particular precondition when violated from distinct call sites should arguably be considered a distinct defect, so counting is benefited by counting distinct caller/callee separately.
- Multiple checks of the same precondition, such as when invoking the same function in a complex expression, can occur on the same line from within the same function, leading to `source_location` not uniquely identifying the defect.

When a defect is not uniquely identified there are two potential fallouts:

1. Diagnosing and fixing the problem can be greatly hindered by needing to narrow down the specific control flow that led to the defect, or worse not even realizing that the defect occurred on a path distinct from the one where a fix was applied.
2. When defects occur on multiple paths, exponential backoff logging can suppress some of them to the point where it can be improperly concluded that there is no defect along the path where diagnostics were not emitted.

Macro-based contract-checking facilities often use a `static` local variable to track the count where the macro is placed, which solves some of these problems but, as with the rest of the macro-based facility's uses, does not allow for capturing the caller location on precondition checks. A contract-violation handler can use the `source_location` object's properties as a key in an associative container to achieve the same level of tracking as this proposal.

A future proposal that makes *observe* a usable contract-checking semantic in the language would be well served to also include the ability to, on a `contract_violation` object, expose a unique identifier for the contract-checking annotation that was violated. The type of this property should be something suitable to use as the key in an associative container while also facilitating easy generation of a unique identifier by the platform. Generating such a unique value can be done by taking the address of the *instructions* that execute the contract_violation process. This would allow for distinct counting of separate inlined versions of the same function in different contexts, which are invariably distinct defects. To facilitate this uniqueness, the type of this property must thus be an integral type at least as large as `std::uintptr_t`.

Keeping an opaque identifier would leave it, like many of the other properties of the `contract_violation` object, up to compiler QoI how accurate and useful the identifier is. Careful wording would need to be constructed to at least guarantee uniqueness as effective as the `source_location` is, but leave more granularity up to the compiler.

5 Throwing

Exception propagation from the evaluation of a contract-checking annotation is, unsurprisingly, in strong conflict with the use of `noexcept`. The Lakos Rule,[12] which proscribed the use of `noexcept` on functions with narrow contracts, i.e., those having one or more preconditions, was invented and applied to all of the C++11 Standard Library precisely to address this issue. Once such an exception escaping from a violation handler begins to propagate up the call chain, any intervening `noexcept` function that fails to catch the exception will force termination. We provide no general solution here to the problems that arise when a throwing handler is invoked for a failing contract check in a `noexcept(true)` function — irrespective of whether it resides on the declaration or within the body of such a function.

When the evaluation of a precondition on a `noexcept` function causes an exception to be thrown, we *do* have a choice as to whether to consider that exception to be happening prior to the invocation of the function or, as is more commonly represented, as the first operation of the function's body. If we choose to define the preconditions as being outside the purview of the function itself and thus not subject to the `noexcept` guarantee, we must then answer a fundamental question about what the `noexcept` operator does when applied to such a function.[13] Almost any answer we can think of would be surprising in one way or another.

- Return `false` in build modes where a precondition check may be evaluated.
- Return `true` or `false` depending on the (link-time) `noexcept` property of the installed violation handler.
- Return `true` yet allow the exception to propagate, breaking some ability to reason about the exception safety of the invoking code.
- Return `false` independently of the build mode, even though the function itself is marked `noexcept`.

All of these concerns fairly demand careful consideration prior to making a decision on how contract checks relate to the `noexcept` boundary of a function.[14]

For functions that violate the Lakos rule, one's ability to either (1) test their precondition checks using the negative testing algorithm described above (see Section 4.7) or (2) attempt to recover using any form of exception-based contract-violation handling (see Section 4.3) is severely hampered by any choice that does not allow exceptions emitted by a contract-violation handler to propagate from the invocation of a `noexcept` function. Therefore, we still strongly recommend that the Lakos rule be followed for any *narrow-contract* functions that do not throw, as doing otherwise actively conflicts with several of the use cases we have presented here.

[12] See [N3279].

[13] The same answer should also coincide with whether the type of the function itself retains the `noexcept(true)` qualifier. This can impact, for example, the ability to assign the address of the function to a function pointer with a `noexcept` function type, as well as the type that would be deduced from the function when it is used as input into template argument deduction.

[14] Because some of these options lie, and others result in fundamental changes to control flow and program behavior outside of the contract-checking itself, we strongly recommend that all checking of preconditions and postconditions remain within the purview of the `noexcept` guarantee of a function, and will elaborate on that in a followup paper — [P2834R0].

6 Wording Changes

The current MVP[15] does not contain suggested wording, somewhat by design. A previous paper, [P2388R4], contains standard wording for an earlier iteration of the MVP, and the final wording for the MVP can be expected to evolve from that.

In [dcl.correct.test], introduced in [P2388R4], add a new paragraph after paragraph 2:

> The *contract-violation handler* of a program is a function of type "*opt* `[[noreturn]]` *opt* `noexcept` function of (lvalue reference to `const std::contracts::contract_violation`) returning `void`" named `::handle_contract_violation`. Whether the contract-violation handler is replaceable is implementation defined. (A C++ program may define a function with this name and signature and thereby displace the default version defined by the implementation.) [*Note:* The definition of a contract-violation handler on an implementation where the contract-violation handler is not replaceable will result in multiple definitions of the contract-violation handler and thus be ill formed. — *end note*]
>
> *Recommended practice:* The default contract-violation handler provided by the implementation should produce diagnostic output that suitably formats the most relevant contents of the `std::contracts::contract_violation` object and then return normally.

Note: The existing wording in [P2388R4] already references the contract-violation handler without further detail, specifying that it is invoked for an unsuccessful *enforced correctness annotation test*.

Add a new section, [support.contract], after section [support.coroutine]:

> **Contract-violation handling** **[support.contract]**
>
> **Header `<contract>` synopsis** **[contract.syn]**
>
> The header `<contract>` defines a type for reporting information about contract violations generated by the implementation.
>
> ```
> namespace std::contracts {
>   enum class contract_violation_detection_mode : *unspecified*;
>   enum class contract_semantic : *unspecified*;
>   enum class contract_kind : *unspecified*;
>   class contract_violation;
> }
> ```
>
> **Enum class `contract_violation_detection_mode`** **[support.contract.violation.detection.mode]**
>
> Enum class `contract_violation_detection_mode` [tab:support.contract.violation.detection.mode]

[15] See [P2521R3].

Name	Meaning
`unknown`	Unknown reason for violation handler invocation
`predicate_false`	Contract predicate returned `false`
`predicate_exception`	Unhandled exception evaluating contract predicate
`predicate_undefined_behavior`	Contract predicate would have undefined behavior when evaluated

Enum class `contract_semantic` [support.contract.semantic]

Enum class `contract_semantic` [tab:support.contract.semantic]

Name	Meaning
`enforce`	End program on violation

Enum class `contract_kind` [support.contract.kind]

Enum class `contract_kind` [tab:support.contract.kind]

Name	Meaning
`pre`	A `[[pre]]` contract annotation
`post`	A `[[post]]` contract annotation
`assert`	An `[[assert]]` contract annotation

Class `contract_violation` [support.contract.cviol]

```
namespace std::contracts {
  class contract_violation {
  public:
    const char* comment() const noexcept;
    contract_violation_detection_mode detection_mode() const noexcept;
    contract_kind kind() const noexcept;
    source_location location() const noexcept;
    contract_semantic semantic() const noexcept;
  };
}
```

The class `contract_violation` describes information about a contract violation generated by the implementation.

```
const char* comment() const noexcept;
```

Returns: Implementation-defined text describing the predicate of the violated contract.

```
contract_violation_detection_mode detection_mode() const noexcept;
```

Returns: The manner in which this contract violation was detected.

```
contract_kind kind() const noexcept;
```

Returns: The kind of contract annotation whose check detected this contract violation.

```
source_location location() const noexcept;
```

Returns: The implementation-defined source code location where this contract violation was detected.

```
contract_semantic semantic() const noexcept;
```

Returns: The runtime semantic chosen (at build time) for the contract annotation that has been violated.

7 Conclusion

With growing concerns over the MVP's severely limited ability to meet the needs of many existing C++ users,[16] SG21 will inevitably be compelled to consider various proposals to address each of those individual concerns. The well-proven approach of supporting a user-defined contract-violation handler has been shown to address these use cases clearly, effectively, and without the need for excessive core-language specification efforts. Although a contract-violation handler does not solve all problems, the new information about the expectations of the MVP clearly indicates that we should reconsider this flexible solution, which would immediately unleash real-world use of the language feature SG21 is striving to produce.

Acknowledgements

Thanks to John Lakos, Bjarne Stroustrup, Tom Honermann, Andrzej Krzemieński, Ville Voutilainen, and Gašper Ažman for feedback on the earlier revisions of this paper.

Bibliography

[bde14] "Basic Development Environment". Bloomberg
https://github.com/bloomberg/bde/

[N3279] A. Meredith and J. Lakos, "Conservative use of noexcept in the Library", 2011
http://wg21.link/N3279

[N4378] John Lakos, Nathan Myers, Alexei Zakharov, and Alexander Beels, "Language Support for Contract Assertions", 2015
http://wg21.link/N4378

[P0465R0] Lisa Lippincott, "Procedural Function Interfaces", 2016
http://wg21.link/P0465R0

[16] See [P2698R0].

[P0542R5] J. Daniel Garcia, "Support for contract based programming in C++", 2018
http://wg21.link/P0542R5

[P1332R0] Joshua Berne, Nathan Burgers, Hyman Rosen, and John Lakos, "Contract Checking in C++: A (long-term) Road Map", 2018
http://wg21.link/P1332R0

[P1429R3] Joshua Berne and John Lakos, "Contracts That Work", 2019
http://wg21.link/P1429R3

[P1607R1] Joshua Berne, Jeff Snyder, and Ryan McDougall, "Minimizing Contracts", 2019
http://wg21.link/P1607R1

[P1639R0] Corentin Jabot, "Unifying source_location and contract_violation", 2019
http://wg21.link/P1639R0

[P2388R4] Andrzej Krzemieński and Gašper Ažman, "Minimum Contract Support: either No_-eval or Eval_and_abort", 2021
http://wg21.link/P2388R4

[P2521R3] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum, "Contract support – Record of SG21 consensus", 2023
http://wg21.link/P2521R3

[P2698R0] Bjarne Stroustrup, "Unconditional termination is a serious problem", 2022
http://wg21.link/P2698R0

[P2751R0] Joshua Berne, "Evaluation of Checked Contracts", 2023
http://wg21.link/P2751R0

[P2784R0] Andrzej Krzemieński, "Not halting the program after detected contract violation", 2023
http://wg21.link/P2784R0

[P2834R0] Joshua Berne and John Lakos, "Semantic Stability Across Contract-Checking Build Modes", 2023
http://wg21.link/P2834R0